doi: 10.7250/tcc.2014.006
Technologies of Computer Control
2014 / 15

Topological Modeling Based Diagnostic Tests Selection

Matiss Erins, *Riga Technical University*

***Abstract*—This article covers the process of software testing. Test management and creation methods are described within the scope of the research. The process of test selection through several stages of project development is discussed and practical examples of appliance are given for the test organization and decision making with the help of topological models of software. The criteria of test ranging are described within scope of each of the testing levels. The paper indicates the use of topological structural models in software test creation, and planning.**

***Keywords*—Decision making, software testing, topology.**

I. Introduction

Effective quality assurance systems are the key to a successful manufacturing. Unacceptable product quality leads to rapid decrease of market demand [1] as it has critical role in fields like medicine transportation, energy, nuclear engineering, where the systems must correspond to the highest standards of quality. Product testing is the basic function of quality support. Software testing is inspection with the objective to collect information about the quality of product under the test.

This research is directed to the software testing phase and more precisely to the selection and ranging of pre–made tests. There are many methods of manual and automated creation of tests [2], [3] discussed in the literature.

This work describes and categorizes the process of test task organization and proposes methods for test selection to systematically choose corresponding subsets of the whole test set. The process of selection is based on software topology and evaluation criteria. The main objective is to use the structural graphs and topological characteristics for test evaluation.

In every phase of software development there is a number of artefacts acquired for the evaluation of next phase [4]. Testing takes different forms depending on software development methodologies [5]. There are various structural control measures like design, quality, test and data measures [6]. The defects discovered during the testing can be traced using methodologies described in [7] and [8].

II. Usage of Topological Model in Test Creation

The tests are generated for a sample program called "Triangle problem" – the algorithm that uses numerical input values for triangle sides and determines the type of triangle. The algorithm is widely used as an example and therefore is extended by additional functionality for test analysis and generation. A full test creation process is described in [9], not all of it used in this research. The topological structural model of triangle program is shown in Fig. 1. The 4 graphs are paths of program flow generated using base path analyses method (McCabe, 1982). Each of the paths describe program path with a different result. Other paths are compared with four base paths for functional redundancy analysis in order to reduce the total test set.

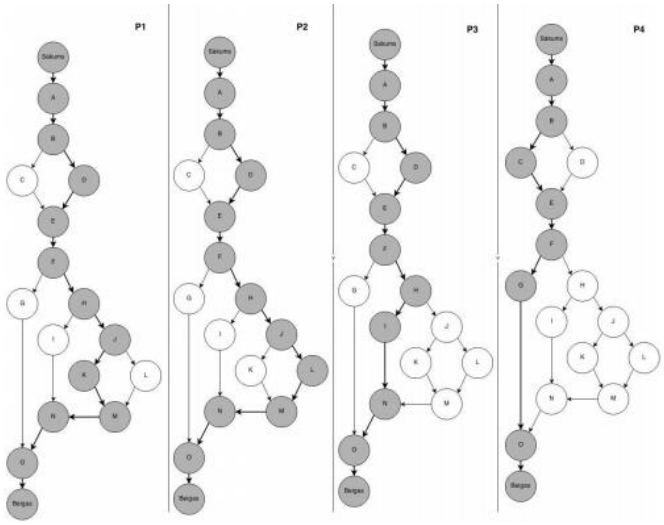

Fig. 1. Program flow topology represented with four base paths.

III. Proposed Test Selection Process During the Testing

The decision making for test selection is split in four levels matching to the stages of software development. At first, for the purpose of component testing, used stages and decision making schemes are taken into consideration. These are followed by the integration stage or inter-component cooperation tests. When all components are developed, it is possible to accomplish system tests and then to do usability testing.

A. Component Tests

Computer system consists of basic building blocks – modules. Modules are created based on structural models, which can be created based on every module algorithm. As the model is the smallest unit *t* in the described stages, it can also be viewed as a single function or multiple functions depending on the complexity. The algorithm of a module can be split into sub–modules which can be combined for view–ability. By assuming that each module is assigned to a single functionality, modules can be named by functional meaning [10]. It is the assumption that those tests to which this module is assigned, belong to the same component. In this level the

TABLE I

INITIAL COMPONENT TESTS AFTER CREATION

Test Number	Structural coverage	Method of test creation	Input values	Expected result	Actual result of last execution	Last execution date ($T_{\_ped}$)	Last execution time ($t_{\_izp}$)
1	0.25	Decision path	a = 3 b = 4 c = 5	Arbitrary	OK	12.12.2013	00:00:13
2	0.25	Border value analysis	a = 3 b = 3 c = 2	Isosceles	NOK	12.12.2013	00:00:12
3	0.25	Equivalence partitioning	a = 3 b = 3 c = 3	Equilateral	NOK	12.12.2013	00:00:12
4	0.25	Worst case	a = 3 b = 3 c = 0	Not a triangle	OK	12.12.2013	00:00:13

software structural graph is used in order to assess the test coverage criteria. Graph structure is useful in analysis in the reflection of logical flow, order of passing the inter-component control and change of variable values. Structural topological model at component level is the base for creation of software tests, based on which higher level tests can be performed.

When the initial set of tests is created, every generated test of the set has determined values of:

- Structural coverage of test, k_p;
- Affiliation name of the component;
- Time of the test creation.

Initial tests created for the triangle problem can be seen in Table I.

The purpose for execution of tests in this level is quality assessment and reduction of test redundancy – providing structural coverage and detection of potential faults. The decision making is based on the coverage of structural graph, weighted particular redundancy measure *NR*.

Functional test redundancy is detected by comparing the purpose of each test. If the values match, the tests are redundant. Structural test redundancy is compared by the path which this test takes in the program graph. If the same path is executed repeatedly, the possibility for redundancy is higher. The correctness of each test step depends on precise values of function weight rating and of the precise definition of functions description. Before initial test execution there is no dynamical statistics of the test. The evaluation is possible by using the tested componential coverage criterion $k_{k.p}$ for each test. This criterion is described by (1), where C_i is the chosen coverage method and NR is test redundancy measure [10]:

$$k_{k.p} = C_i\, NR \tag{1}$$

The tests are ranged by structural coverage criterion $k_{k.p.}$ The whole test set needs to be executed to reach the structural coverage $t_{\ strukt}$ to be as close as 100 %. If all of the paths are not reachable, then kvazi–optimal structural coverage is below this value.

After the ranging, tests are added to the executable test set. The test set is assigned the total structural coverage value C_k by cyclic addition of set results shown in (2), where C_i is coverage of the current test:

$$C_k = C_k + C \tag{2}$$

Test addition to the set is continued until the condition of sufficient percentage condition is reached (3), where N_{strukt} is number of elements chosen by the criterion:

$$\tau_{strukt} \geq \frac{C_k}{N_{strukt}} * 100 \tag{3}$$

After the execution of the second phase the following test dynamic criteria are acquired (3), where :

- Test execution time t_{izp};
- Last test execution date T_{ped};
- Test failure statistics (test passed or not).

After the execution of tests it is possible to evaluate the test fail statistics for the component under the test. The failed tests are then analyzed and rated by importance and priority (Table II).

TABLE II

TEST SET AFTER EXECUTION

Test Number	Structural coverage, *kp*	Method of test creation	Input values	Expected result
1.	0.25	Decision path	a = 3 b = 4 c = 5	Arbitrary
2.	0.25	Border value analysis	a = 3 b = 3 c = 2	Isosceles
3.	0.25	Equivalence partitioning	a = 3 b = 3 c = 3	Equilateral
4.	0.25	Worst case	a = 3 b = 3 c = 0	Not a triangle

The aim of the repeated testing is to test if the failures reported previously have been fixed. Decision is made based on the composite value for the set of dynamic criteria. Repeated testing executes the tests that were marked as "NOK" in previous executions. Decision is made based on the following parameters (4):

$$k_{def} = \omega_s k_{p,} \quad (4)$$

where

ω_s defect weighted severity ;

k_p defect priority.

The tests are ranged by the dynamic criterion. After several cyclic re–testings the last result is accepted. Test management tools like Testia Tarantula can store up to 3 last execution results for each test. The repeated testing set contains only the tests with the last failed execution.

B. Integration Tests

Integration testing is the phase when separate modules of software are tested in groups. It is done before the system tests. Integration tests consist of unit tests with already valuated criteria, which are valued at the component testing phase. Integration tests are added by specified integration test plan. Unit tests are combined and added or assigned to these integration tests.

Components are nodes of component relation graph. Relational graph can then be condensed to a single node of integration graph. Edges between the nodes of the integration graph connect the output of the structural node to the input of the continued structure in structural level. When the number of graph nodes reach 100 it is suggested by structural modelling to use graph condensation. In case of the systemic structural graph the cyclic structure is too complex, the structure is scaled [11]. The order of setting the test steps in integration tests follows by the component relations in oriented graph.

Statement: the edge created between nodes does not have a role of input or output signal or exposure. It points out that there exists cause and effect relation between these nodes and reflects binary relations in the set of functional properties. [11]

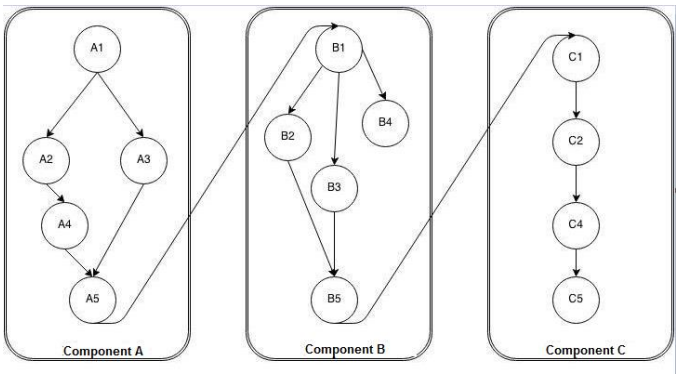

Fig. 2. Program graph splitting components.

Program decision graph (Fig. 2) is separated for testing into components which are developed and gradually integrated to real project. The following approach facilitates project development and creation of tests. Parts of a graph with the same assigned functionality are grouped under the same functional node, but edges in Fig. 2, like {A5, B1} and {B5, C1} are turned into edges of the integration graph {A, B} and {B, C} (Fig. 3). For oriented graph the successor set of a node consists of nodes to which this is the input edge. For the same node the predecessor set is formed by nodes to which this node is outgoing edge apex.

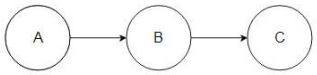

Fig. 3. Functional condensation graph of a structural model.

Integration phase input conditions:

- Project code has passed component testing phase;
- Product satisfies the requirements of performance and memory detailed in functional test specification;
- Software has passed tests for basic evaluation of failures;
- All component level high priority issues are fixed;
- Documentation is updated to correspond to the current status of the project.

There are following strategies in integration phase discussed within the scope of this research – top-down and bottom–up integration.

With the top-down approach the main module of software is tested at first, for other modules there are specific drivers created. Then planwise the drivers are replaced with actual component with drivers (Fig. 4). This is done until there are no other modules called. It is important for the first test modules which use interfaces, I/O operations and modules which have the highest failure rate [12].

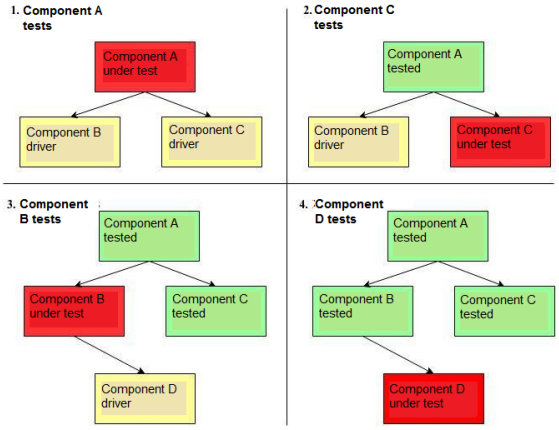

Fig. 4. Top-down testing.

The benefits of the top–down approach are: 1) Reduced time for system tests; 2) Highest level interfaces are tested first while testing also the lowest levels. During these tests most failures are localized in the last recently added modules. The disadvantages of the method are: 1) Need to create specific drivers; 2) It is relatively hard to find test data for new modules. The testing data flow is also including non–oriented graph and the testing of interface is costly.

The bottom–up testing approach takes a module which does not call other modules. At first this module passes the component testing phase and the testing is concluded for the modules calling already the tested ones i.e. interface between subsystems. Tests can be executed for multiple groups in parallel (Fig. 5). Simple test data can be created for smaller modules, but the problem lies in the complexity of input or overall environment simulation. In the case of many modules, large number of subsystems will be tested at the same time.

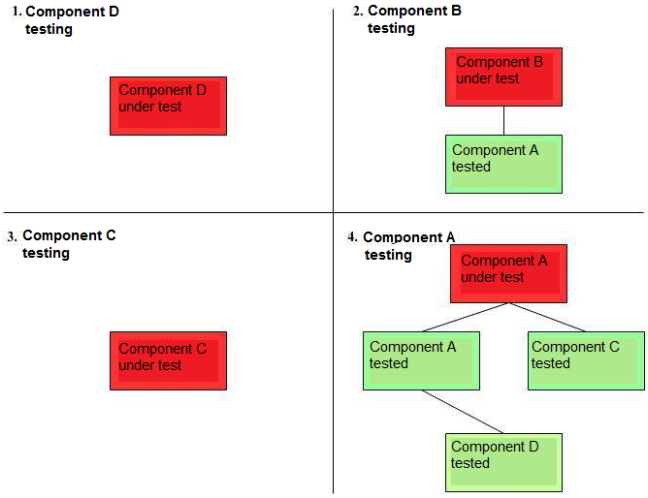

Fig. 5. Bottom–up testing.

To calculate the number of integration tests for the component (5) is used, where V is the cyclomatic complexity:

$$V = \sum_{i-1}^{N} v_i \tag{5}$$

The total number of integration tests for the 3 components used in tests is the sum of tests for each component in the test set (6).

$$V = (A) + (B) + (C) \tag{6}$$

The weight is each step distance in graph. The criterion is the sum of sub-graph coverage or the sum of the component coverage. All pair paths are a subset of paths that are combined from sub path nodes. Sub path and sub path set is characterized by the relation of many-to-one. If sub path includes a loop, the sub path associated with the set does not affect the change in the number of loop iterations. The set of pair path is the set of edges that includes pair paths. For each x in test T the set of exercised paths includes all pair path sets that are defined for all reachable users. If a path involves loops, all path testing requires two test tasks, the first does not involve passing loop elements, the second executes the loop element for a given number of times.

Phase 1: Test base sets are chosen based on the structural parameter of each module (Fig. 6).

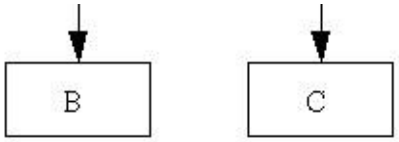

Fig. 6. Separate test creation for sub–modules B and C.

Phase 2: Test base is created and selected for the edges connecting the main module A with sub–modules B and C (Fig. 7).

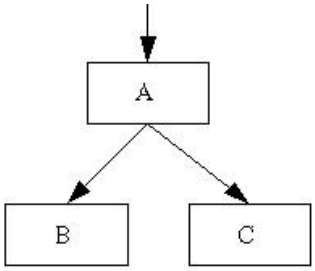

Fig. 7. Integration of module A with modules B and C.

The complexity of integration $S1$ is defined for a program with n number of modules (G_1 *to* G_n) by using equation (7):

$$S1 = (SUM\ iv(Gi)) - n + 1 \tag{7}$$

The complexity of integration measures the number of independent integration tests in the scope of full software design. In the integration level test tasks for separate modules are combined by insertion into a single test task using the principle that the last step of the first task is the first step of the second task, in this way creating the integration level of test steps from test tasks in component level. The single test step is defined by the ID – identifier. By looking at the test creation for the triangle problem (Fig. 8), the program is split into three components A, B and C. These are modules which are integrated step–by–step into the solution and the integration tests are made.

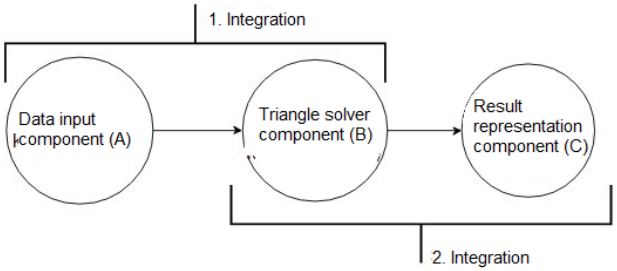

Fig. 8. Integration level graph.

For the execution of the first integration stage there is a test which contains merged A and B components (Table III) as well as the Test 2 for the second integration.

TABLE III

INTEGRATION LEVEL TEST EXAMPLES

Integration test 1			
Step ID	Component	Action	Result
1	A	Input a = 3, b = 3, c = 2	Data acceptable
2	B	Input a = 3, b = 3, c = 2	Isosceles
Integration test 2			
Step ID	Component	Action	Result
1	A	Input a = 3, b = 3, c = 2	Data acceptable
2	B	Calculate a = 3, b = 3, c = 2	Isosceles
3	C	Triangle type and sides a = 3, b = 3, c = 2.	Image of isosceles triangle

The execution of integration level tests requires evaluation of structural coverage for each test. The distance between the two components is the length of path or the number of edges in the integration graph. The weight for this criterion w_{att} is expressed in (8):

$$w_{att} = \frac{1}{d}. \tag{8}$$

The coverage measure of test step integration is calculated using the weighted component distance measure. The weight indicates (9), the order of components compared:

$$k_{i.p} = w_{att}\ k_{k.p}\ . \tag{9}$$

The total functional test value for integration level is the weighted sum of individual steps (10). The value is weighted by the rate of the step included in the set:

$$w_b = \frac{N_{ID}}{N_{S_{ID}} * 100}\ . \tag{10}$$

The equation for k_t integration test value (10):

$$k_t = \sum_{N=0}^{N_s} k_{i.p} w_b. \tag{11}$$

The tests are put based on test severity criterion k_t in descending order by ranking higher tests with a higher value of k_t. (Table IV).

TABLE IV

TEST OVERVIEW

	Integration test 1					
Step ID	Component	w_{att}	$k_{k.p}$	$k_{i.p}$	w_b	k_t
1	A	1	0.141	0.141	0.165	0.0235
2	B	1	0.25	0.25	0.02	

	Integration test 2					
Step ID	Component	w_{att}	$k_{k.p}$	$k_{i.p}$	w_b	k_t
1	A	0.5	0.141	0.07	0.16	0.0335
2	B	1	0.25	0.25	0.02	
3	C	1	0.4	0.4	0.025	

The exit conditions for the integration phase are as follows:

- Component setup tests are done;
- All priority defects are fixed and closed;
- Earlier documentation is updated to match the current condition.

C. System Tests

During the system testing the behavior of system or product is tested against the expected behavior stated in the documentation. It is possible to include tests based on risk or requirements specification, business processes, use cases or other high level description of system functions and interaction with operating system or system resources. System testing is chosen as the last phase to gain confidence that the object under the test corresponds to the specification. System testing is held by test specialists of independent testers. System testing is intended to check both the functional and the structural requirements by focusing on the functional side [13].

System tests are formed from integration phase tests and component tests by using the requirement traceability matrix. Requirement traceability matrix is a document with the many-to-many correlation of two documents (Table V). This approach is used for the system test creation, by adding user requirements to tests and the evaluation of quantity percentage $k_{\text{req.}}$

TABLE V

EXAMPLE OF TRACEABILITY MATRIX

Test task	Requirements	P1	P2	P3	*k*req
	Total	2	2	1	
*T*1	2	X	X		40 %
*T*2	2		X	X	40 %
*T*3	1	X			20 %

Table V consists of three tests and 3 requirements bound together. Each test has the percentage evaluation of requirements covered by the test and a number of tests that cover the requirement. System tests are executed in a similar way to integration tests where the criterion of requirement coverage k_{req} is used as a measure.

Rating = Structural coverage + requirements coverage

The value of requirement coverage is calculated by the ease of selected system test. System tests are given values of selected weight *wreq*, which depends on the volume of the selected test set. This paper describes 3 test set volumes – a

small volume test called “smoke test”, a requirement volume and a full test set.

The small test set idea is to check the normal function of the main system components. If these tests are repeated, it is possible to assign tests with higher severity for a repeated testing in order to check for last changes and they are characterized by the highest fault possibility. The purpose of the easy test set is to call each main function of the system by taking into account the time constraints.

The requirement tests depend on the requirement coverage. Requirement coverage of 100 % means that the set tests all requirements are assigned. Requirements can be visualized in program integration graph by adding requirement description for each component. Fig. 8 contains the component nodes created in lower level tests with attached requirement description with the chosen level of detail, which are then associated with tests.

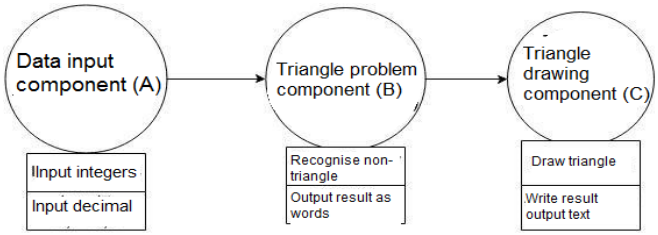

Fig. 9. Component level graph with added requirements.

Before system activation or launching into production environment a full system test is executed. In the case of the full test set tests are organized by the test coverage structural criterion combined with the dynamic criterion of test execution time. The time constraint is regulated by sorting tests in ascending order, where first executed are the tests with shorter average duration and higher structural coverage having the longest execution period. The exit criteria for the system tests are:

- Requirement coverage for tests has reached a set value;
- All defects with high or average severity are fixed;
- Software is tested to work with all supported devices, system configurations and other products [14].

D. Usability Tests

When the tested system is passed to the real user or client usability tests are made. The purpose of testing of the application is to make sure that the feature or use case is included in the system and used the proper way. The test selection is similar to the integration stage, except that the tests are made by focusing on user stories not on actual components [15].

Usability tests are executed when it is possible to measure the ability for a system or a subsystem to respond to the requirement specification usually after the implementation of larger project parts or system versions. During this phase, new tests are generated based on sequential diagrams. Also there are automated tools for test generation [16] using principle of usage cases. This paper is focused on the tests created during previous phases and on the customization of tests in usage level. Entry criteria for usability phase are as follows:

- Usability test plan is confirmed;
- All high priority system level issues are solved and defects fixed;
- Software is capable of working on all supported devices and platforms [17].

Usability level tests consist of user steps – sequential actions which must be executed using the software under the test. These tests check possible user roles and access to the role-specific systemic functions. Usability tests can be done in the field of security testing by means of checking boundaries set in software. Structural topological models of system can be applied for usability test creation when component tests are combined and the steps of usability tests are then matched with the nodes of integration graph the decision making during this phase is based on system level requirements. Usage coverage criterion is used for test ranging. Output criteria for the usability testing stage are:

- Usability tests should reach the threshold of usability coverage (i.e. 80 % of tests done);
- All defects of usage marked as “high priority” should be tested and fixed.

IV. Results

Organizational methods and structural approach to the test selection is the basic instrument for automated test selection and creation and execution. The test management software structure was discussed with possible solutions to access and acquire test data form data base without affecting the test management structure. Theoretical approach of test organization with given practical examples is given in this paper. Tests can be organized on different levels matching the stages of project development. Independent test analysis can be held in any of the four stages described and the results are used for higher level test organization. Sorting criteria are defined for use in each of the levels. The objective of the research is the creation of automatic test selection tool and shared database for tests. The possible solution is shown in Fig. 9. The test tool integration is intended to have a user level access to the tests stored by the test management tool. The proposed test processing module is based on the decision making and test selection block that uses methods described in this work. As the testing environment of “Testia Tarantula” management tool supports agile software development methodology as well as the methods for current purpose support step wise integration, the planned solution would be used in this project.

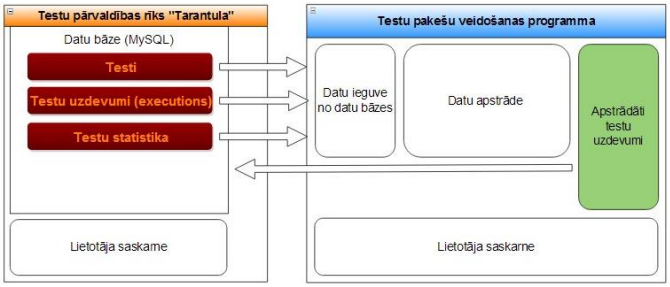

Fig. 10. Test packet selection tool integration.

The test management tool “Testia tarantula” is an open-source project which supports bug tracking features and other software tools like “Jira” and “Atlassian” [18]. This solution supports agile software development methodology with multi-user support. The structure of the test data base can be accessed by any “MySQL” database manager. The management solution is built on Ruby programming language.

The concept of the automated test selection tool is to display the test set using the structural graph of software which can be scaled by the selected stage of the development to facilitate the selection of test set. The output data for the proposed module consist of a ranged test set that is updated before the test execution. The implementation of the described methods into an automated tool minimizes the manual test redundancy and improves efficiency of the regression test set, requiring more detailed research and usage statistics. The solution can be used for small to medium scale projects where the structure of a single component or systemic component does not exceed 100 to 1000 units. Larger scale projects would require separation of low level components. Decision paths can be modelled by using structural models created in test selection. Structural and functional models are used to graphically analyze software structure and to evaluate the impact of the selected test set. Usage of structural graphs in functional testing can indicate the functional redundancy.

REFERENCES

[1] American Society for Quality, Glossary for word “Quality”, [Online]. Available: http://asq.org/glossary/q.html [Accessed Nov. 6, 2014].

[2] R. Nilsson, “Automated Selective Test Case Generation Methods for Real-Time Systems,” M.Sc. thesis, Comp. Sci., Univ. of Skovde, 2000. [Online]. Available: http://www.diva-portal.org/smash/get/diva2:2866/FULLTEXT02 [Accessed Nov. 6, 2014].

[3] J. Rushby, “Automated Test Generation and Verified Software,” *SRI International,* [Online]. Available: http://vstte.inf.ethz.ch/Files/rushby.pdf [Accessed Nov. 6, 2014].

[4] ISTQB Foundation, Software Development Life Cycle phases. [Online]. Available: http://istqbexamcertification.com/what-are-the-softwaredevelopment- life-cycle-phases/ [Accessed Nov. 6, 2014].

[5] C. Larman. *Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 1st ed., ISBN 9780131111554, pp. 9–17.

[6] S. H. Kan, “Software Quality Metrics Overview,” in *Metrics and Models in Software Quality Engineering*, 2nd ed. ch. 4. pp. 85–120. [Online]. Available: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0201729156.pdf [Accessed Nov. 6, 2014].

[7] ISTQB, “Certified Tester Advanced Level Syllabus: Test Analyst,” p. 64, 2012. [Online]. Available: http://www.istqb.org/downloads/finish/46/95.html [Accessed Nov. 6, 2014].

[8] H. Stone, *Software Metrics Proposal*, 1998. [Online]. Available: http://softwaretestingservice.com/SoftwareMetricsProposal.pdf [Accessed Nov. 6, 2014].

[9] R. S. Pressman. *Software engineering: a practitioner’s approach,* 4th ed., McGraw-Hill, Inc. 1997, pp. 852.

[10] Agile alliance. *Unit testing*. [Online]. Available: http://guide.agilealliance.org/guide/unittest.html [Accessed Nov. 6, 2014].

[11] J. Osis, J. Grundspeņķis, Z. Markovičs, *Topological Modeling of Complex Heterogeneous Systems: Theory and Applications*. Rīga: RTU, 2012. 407 p. ISBN 9789934507014.

[12] S. Anderson, School of Informatics. Integration Testing [Online]. Available: http://www.inf.ed.ac.uk/teaching/courses/st/2011-12/Resource-folder/10_integration.pdf [Accessed Nov. 6, 2014].

[13] ISTQB, “Certification – What is system testing?” [Online]. Available: http://istqbexamcertification.com/what-is-systemtesting/ [Accessed May 19, 2014].

[14] R. Vivek, Entry and Exit Criteria for Different Stages of Testing. [Online]. Available: http://vivekranjan1980.wordpress.com/2010/03/23/entry-and-exit-criteria-for-different-stages-of-testing/ [Accessed May 19, 2014].

[15] L. Luo, “Software testing techniques. Technology Maturation and Research Strategy,” [Online]. Available: http://www.allbookez.com/pdf/14d8it/ [Accessed May 19, 2014].

[16] S. Kariyuki, H. Washizaki, et. al., “Acceptance Testing based on Relationships among Use Cases,” *5th World Congress for Software Quality*, p. 25, 2011. [Online]. Available: http://www.juse.or.jp/software/390/attachs/paper04.pdf [Accessed Nov. 6, 2014].

[17] ISTQB, “Advanced Level Syllabus. Test Analyst,” p. 64, 2012. [Online]. Available: http://www.istqb.org/downloads/finish/46/95.html [Accessed Nov. 6, 2014].

[18] Tarantula. (2014) *Test management tool “Testia Tarantula” manual*. [Online]. Available: http://www.testiatarantula.com/ [Accessed Nov. 6, 2014].

Matiss Erins received the degree of *B. sc. ing.* in 2012 and the degree of *Mg. sc. ing.* in 2014 from Riga Technical University.
He is a PhD student with the Faculty of Computer Science and Information Technology, Riga Technical University.
His research interests are: mobile software development, embedded hardware and robot control systems.
E-mail: matisserins@rtu.lv

Matīss Eriņš. Topoloģiskajā modelēšanā balstīta diagnostisko testu izvēle
Pētījumā sīkāk apskatīta testu uzdevumu pārvaldība un testu uzdevumu veidošanas metodes. Teorētiskie pamati satur informāciju par programmatūras izstrādes procesu kopumā – izstrādes posmi un metodoloģija, tāpat detalizēti apskatīts testēšanas process un tā iekļaušana izstrādes dzīves ciklā. Darbā apkopoti programmatūras izstrādes procesā izmantojamie kritēriji. Pētījumā analizēta diagnostisko testu atlases lēmumu pieņemšanas gaita dažādos programmatūras integrācijas līmeņos, kā arī apskatīta grafu modeļu izmantošana testu izveidē un plānošanā. Apskatītas iespējas ar grafu īpašību palīdzību samazināt testu atkārtošanos un noteikt testu atlases kritērijus atšķirīgiem testēšanas veidiem. Darba aktualitāte tiek pievērsta šobrīd programmatūras izstrādē aktuālajai spējās izstrādes metodoloģijai un testu procesa iekļaušanai tajā. Darba mērķis ir apskatīt testu uzdevumu organizēšanu un veikt testu apakškopas atlasi, izmantojot uzdotos kritērijus un atlases metodes, balstoties uz programmatūras topoloģiju. Citiem vārdiem sakot, izpētīt iespējas strukturālo grafu īpašību izmantošanai testu izveidē un testu kopas izlases novērtējumu iegūšanā. Darbā tiek aplūkoti esošu testu uzdevumu pārvaldības rīki, veidojot to salīdzinājumu. Uzmanība pievērsta bezmaksas rīkam “Testia Tarantula” manuālai testēšanai.

Матисс Ериньш. Выбор диагностических тестов на основе топологического моделирования.
В исследовании детально рассмотрены методы управления тестовыми заданиями и формирования тестовых заданий. Теоретическая основа работы содержит информацию о процессе разработки программ в целом – этапы и разнообразие методологии, а также детально рассмотрен процесс тестирования и включения ее в жизненный цикл разработки. В работе представлены критерии программы, используемой в процессе разработки . В исследовании проанализирован отбор диагностических тестов в ходе принятия решений на различных уровнях программной интеграции, а также рассмотрено использование графической модели в создании и планировании теста. Обсуждаются варианты уменьшения числа повторений испытаний и тестов с помощью свойств графа, чтобы определить критерии отбора тестов для различных видов тестирования. В рамках работы исследована функциональная возможность дополнения и структурного анализа теста управленческой среды «Tarantula» для создания дополнительного инструмента выбора тестов на основе результатов исследования.